

Creating Administrative Tools

Operating system vendors and network administrators have had a running feud since the computer was invented. It always seems as if the vendor is more interested in introducing new gizmos rather than in the tools that the administrator sees a need for today. On the other hand, administrators have always wanted tools that addressed specific needs rather than look to see if the latest gizmo could somehow answer an existing need.

No matter how many books get written, tools get designed, and gizmos get added, some administrator somewhere won't have the tool needed to get some special job on the network completed in a reasonable amount of time. One of the main problems is that every company is slightly different. Not only do the hardware and software differ from company to company, but business needs, political climate, and expected results differ as well.

That's the reason for this chapter. There are going to be times when no one will have the tool an administrator needs to get the job done, so a custom tool isn't only warranted, but may represent the only way to get the desired functionality. In essence, this chapter's going to look at something that administrators have been doing for quite some time now: Creating specialty tools that allow them to work faster and with a lot fewer headaches than many off-the-shelf tools.

The first two sections of the chapter look at the various kinds of tools and which works best in a given situation. Sure, all of the tools are used by administrators, and they all deal with computer issues; but just as you wouldn't use a screwdriver to drive a nail into a board, you need to use the right tool to

21 CHAPTER



In This Chapter

Understanding the various administrative tool types

Choosing the right administrative tool to get the job done

Creating administration-specific utilities

Adding an Internet interface to administrative tools

Creating administrative tools that work with MMC

Creating administrative tools with an HTML interface



get the job done. (Obviously, there's going to be some overlap in tool functionality, at which point it becomes a matter of personal preference when deciding which tool to use.)

The next section of the chapter looks at a topic that's becoming a matter of concern for many administrators. With some companies becoming larger and more dispersed, there's a need for tools that can deal with servers from a remote location. However, the method used to create the connection between the administrator and the remote computer is cause for concern. Leased lines require special installation and are expensive; not only is the Internet available, it is also relatively cheap. Obviously, the Internet isn't nearly as secure as a leased line, so security becomes a problem that toolmakers need to deal with.

The three sections that follow discuss three of the tool types that you can create using Visual C++. All of them emphasize different methods to work with Windows 2000 and incorporate varying technologies to get the job done. The main emphasis of these examples is that there's always more than one way to create a tool. The choice of methodology is often more a question of where the tool will be used, what task the administrator wants the tool to perform, the environment in which the tool will be used, and how the administrator intends to use the tool.

**Note**

All of the examples in this chapter use new Windows 2000 features that weren't included in Visual C++ 6. You must download the latest Platform SDK from <http://msdn.microsoft.com/developer/sdk/platform.asp>. After you install the new Platform SDK, you'll be able to compile the examples in this chapter. The examples on disk assume that you've placed the Platform SDK in the C:\Program Files\Microsoft Visual Studio\Platform SDK directory of your hard drive. If you've used some other storage location, then change the locations of these files that are shown on the Directories tab of the Visual C++ Options dialog box (use the Tools | Options command to display this dialog box). To run the examples, you must be using Windows 2000 as your operating system. The examples won't run on older platforms such as Windows NT 4 or Windows 98.

An Overview of Tool Types

There are a lot of ways to classify the tools that administrators commonly use on a network. Deciding what kind of tool is required to complete a given task is an important part of the tool specification. The job that the tool is required to perform affects its appearance—what capabilities the tool possesses.

Classifying by General Use

An obvious method of classifying a tool is to determine what you'll use it for in a general way. For example, do you want a tool to monitor the network and alert you to possible problems, or do you need a proactive tool that performs maintenance on a regular basis so that there won't be any need for monitoring in the first place? The following list talks about some of the ways to define tool type by general usage.

Monitor

This is the type of software that administrators use on a daily basis. It helps them track the current status of the network. For example, monitor software can tell an administrator how much memory the server has left or the current status of the hard drives.

Analysis

There are times when simply knowing the network's statistics doesn't really do everything you need it to do. Analysis software takes raw input from monitoring software, studies it, and then returns the result. For example, an analysis tool might try to predict hardware failures based on certain network trends such as the number of errors found while doing normal hard disk maintenance.

Event

Tools don't always provide some type of proactive feature; they may be reactive instead. A hard drive failure is something that you can prepare for and even anticipate, but it's not something that you can predict with absolute accuracy. As a result, a hard drive failure is an event that you might want to look for using some type of tool.

Automated Maintenance

There are some types of maintenance that you need to perform daily that don't require any form of administrator input. For example, tape backup software has been automated for quite some time. The administrator only needs to set up the parameters once, and then the software can automatically perform this maintenance from that point on. Any time you can automate a maintenance task, it increases network administrator efficiency and reduces the chance of error.

Manual Maintenance

Testing software was probably one of the first administrator tools created. It's important to test the network and verify that it actually works as intended—that there isn't some hidden flaw waiting to cause data corruption or other problems. Some of these testing tasks require constant monitoring, input, or even analysis on the part of the network administrator. These kinds of tasks are perfect candidates for manual maintenance utilities.

Configuration

Software today is more flexible than ever before, which means that there are more configuration options than before. Sometimes an operating system vendor hides some of the more powerful configuration options because they fear that the user will use them incorrectly. This particular problem happens all of the time with Windows. Power Toys, a set of utilities designed by some Microsoft engineers, is a perfect example of a tool designed to overcome the limitations of the built-in configuration options of the operating system.

Installation

Administrators constantly perform installations of all kinds — everything from word processors to new operating systems. As a result, this is one area where even a small amount of automation can produce a large increase in administrator productivity.

User Assistance

All of the large software companies have come to the same conclusion about user support: It's less expensive to write software to do the job than it is to hire support personnel. Witness all of the automated methods these vendors employ to direct your attention to some other source of help. Telephone lines usually have menuing systems that offer everything from fax back support to recorded message help. Online sites provide complex search systems that allow users to find information themselves rather than depend on human support. Once you look at these large vendor offerings and analyze why the vendor is going through all the trouble to set up nonhuman user assistance, it becomes pretty obvious why you might need to do the same thing to make administrators at your company more efficient as well.

Tracking

Records are the mainstays of many administrators. Unfortunately, few administrators have the time to keep records that are both complete and up-to-date. Automating the task of tracking various maintenance and error events on a server makes it easier for an administrator to locate problems based on the history of the system.

It's not hard to see that there are a lot of different tool functions. There are probably a few tool types that this list doesn't even consider. In short, you need to figure out exactly how you plan to use a tool before you begin to design it. The way you use a tool will decide things such as the user interface and the form factor of the tool itself — some automated tools may not require any user interface and you might even place them in a DLL instead of an EXE.

Classifying by Location, User, and Data Sensitivity

Another way to look at tools is where they'll be used and by whom. For example, you'd need a tool with an extensive help system and limited potential for harming the network if the person using it was a novice administrator or trainee. On the other hand, an expert administrator who wanted total server control from a remote location would want a tool with robust features, a minimum of chatter (overly helpful messages and help system), and a small footprint so that the tool would be portable and easily downloaded if necessary. The following list looks at the ways that you could classify tool type by location and user level.

Distance

The distance between your workstation and the server makes a great deal of difference in the type of tool you create. A tool that's used on a LAN doesn't necessarily have to be as small or network bandwidth efficient as one created for use on the Internet. In some cases, this means that a tool designed for LAN use can combine more functions into a single large tool rather than smaller single-use tools that are easy to download.

Level of Expertise

A tool designed for the new administrator will normally require more in the way of assistance features. For example, the tool may include more dialog boxes designed to alert the administrator to potential risks and help screens designed to fully describe tool features. Unfortunately, adding a lot of these types of features can significantly increase the size of the tool and make the job of managing the network a frustrating experience in button pushing for experienced users. As a result, you normally want to create tools that consider the needs of the administrator they're designed to help.

Communication Media

LANs are normally high-speed connections, so creating a tool that communicates with the server a lot isn't a big problem. On the other hand, a dial-up connection is almost ridiculously slow. Obviously, you want to optimize communications for the remote administrator using a dial-up connection. In some cases, this may even mean cutting features in exchange for speed.

Data Sensitivity

Adding security to an application slows it down. It doesn't matter whether that application is an administrator tool or not. Some of the tools you design will handle extremely sensitive data such as user passwords, whereas other tools will handle nonsensitive data such as server processing speeds. In short, the sensitivity of the information the tool is designed to handle determines whether that data needs to be encrypted (as well as the level of encryption), which in turn determines things such as tool-processing speed.

Management Requirements

A tool that simply monitors network statistics is less complex to build than one that uses those statistics to predict things such as hardware failures. The complexity of the management task that the tool is designed to handle determines its characteristics.

Determining Which Tool Type to Use

In some cases, the choice of tool type is relatively easy to make. For example, a sole administrator who works with a small-sized company with three shifts will definitely want a set of tools that work well over a dial-up connection. (Unless, of course, the administrator actually enjoys going to work in the middle of the night only to discover that there's absolutely nothing wrong with the server.) For example, the administrator may want a tool that checks the current level of processor usage or that can determine whether a hardware failure really has occurred. However, some choices aren't quite so easy because you don't know what an administrator will require in the future. For example, just because the administrator performs all network maintenance on the job site today doesn't necessarily mean that this will be true in the future.

Unless the use, location, and expertise level for a tool is well known it pays to spend some time looking at both current and future needs. Custom tools take time to build—longer, in many cases, than custom applications—so assessing the requirements for the tool up front is critical. Part of the tool type determination process, therefore, is to figure out how the administrator will use the tool in the present and in the future.

Obviously, part of the longevity consideration hinges on how long you expect to actually need the tool. A tool designed to answer a shortcoming of one version of an operating system may not be needed for very long. On the other hand, a special monitoring tool may be useful through several server software updates.

Another problem is determining how much and what type of information the administrator actually requires. For example, you may build a monitoring tool that uses existing server data only to find out that the administrator ends up calculating the value they really needed. In some cases, an analysis tool that does some or all of the calculation work for the administrator is actually a much better choice than the easy to build monitoring tool.

There's also a question of how much automation a tool really needs. Too much automation may give the administrator a false sense of security, making it possible that the tool won't actually prevent the catastrophe it's designed to prevent. Automation should relieve an administrator of needless tedium, not the responsibility of making decisions based on all of the available facts.

Defining expertise level is something that can be difficult as well. Even an expert administrator may have educational or experiential gaps that prevent them from getting full use from a custom tool. If portability is a concern, but the administrator's true level of expertise is in question, you might consider making help an optional part of the tool. The administrator could always download just the amount of help he or she needs to do the required work.

As you can see, it's important to not only determine exactly what kind of tool you need to create, but to specify the range of its operation as well. In many cases, custom-built tools start out as a handy gizmo on the programmer's desktop and end up in the hands of a user who may not have any idea of what it does. Designing a tool properly from the outset reduces the risk that a tool will be used improperly, even if it does fall into the hands of someone who was never supposed to see, much less use it. In short, always choose the type of tool based on who will most likely end up using it, rather than the person who originally designed it.

Tip

Password protection is a wonderful thing when it comes to administrative tools. Adding password protection doesn't require much code and can take dangerous tools out of the hands of novice users who could use them to damage the network. Obviously, the kind of security you add depends on just how dangerous the tool is. However, it always pays to err on the side of security rather than to find out later that your tool increased the network administrator's workload after a server failure.

Providing Internet Access to Your Tool

There isn't any doubt that you'll use any tool you create on the LAN to get work done, but many administrators are no longer limited to just working from the office. That's where things such as dial-up connections come into play. You could conceivably use products such as PC Anywhere to create a remote session with your workstation and administer the network that way. However, what happens when work is no longer a local call? At that point, it may be time to look at the Internet solution of a virtual private network (VPN).

Creating a tool that can monitor the server's current temperature or determine whether a hard drive is bad isn't as hard as it used to be. Not only has Microsoft added APIs and various counters to help you out, but some hardware vendors now include the sensors needed to perform some types of monitoring activities as well. As a result, you can get more information about your server than ever before from a remote location.

Of course, getting this information through an Internet connection isn't quite the same as getting it from the server to your workstation on the LAN, but there's a common solution to both problems that will reduce your workload. You can monitor your server from any location if you build the right type of components and install them the right way. All you really need is a counter, a DCOM component, and an application that's built to access the remote component. We looked at the DCOM part of the picture in Chapter 14. In this chapter, we look at building your own performance monitor counter that can be designed to check the status of any part of the hardware as long as you know how to access it. This even includes things such as temperature sensors. (Most vendors who provide temperature sensors with their motherboard and other components also include local software to monitor these sensors.)

Obviously, gaining access to the information you require isn't the only problem. A tool designed for Internet use also has to be small enough to download if necessary. If you plan to use a browser to access the component on the server, a download may very well be necessary. Using ATL-based COM components is probably the method of choice, in this case, because you'll definitely want to create the smallest possible implementation without losing much in the way of component flexibility.



We covered ATL-based component creation in Chapters 12 (theory) and 13 (implementation), so you may want to review those chapters as well if an Internet solution to an administrator tool need is in your future.

Another potential problem for tools that can work on the Internet is security. There are actually three levels of security that you need to worry about. The most obvious problem is unauthorized access. You don't want just anyone to be able to access your server to perform maintenance. Some people might perform maintenance of the most unhealthy sort.

The second security problem comes in the form of data transmission. Someone with a sniffer could learn your company's most sensitive secrets if you don't protect the flow of information from the remote workstation to the server and back. Encryption is one way to take care of the problem, but you'll also want to make sure that the information you transfer from one place to another is kept to a minimum. In other words, you need to design your tool in such a way that it doesn't request any more information than absolutely necessary to get the job done.

The third security problem is the tool itself. Allowing someone to remove an administrative level tool from the work site is one sure way to invite them to take it apart to see how it works. Theoretically, someone could gain quite a bit of knowledge by looking at how a tool is put together and analyzing the methods it uses to garner information from your company server.

Fortunately, Microsoft provides answers to the first two problems as part of Internet Information Server (IIS). You can secure your Web site and require the user to log in using the same methodologies that they use to log in to the local network. Because their name and password is encrypted (at least if you use the Windows 2000-specific security), there's little chance someone else will discover who they are. Once the user is logged in, you can establish a secure connection with them using certificates. This means that the data will be kept secure as well.

The third problem isn't as easy to solve. You have to either trust the employee not to allow someone to see the specialty software, or you need to find some way to allow them access without giving them the control. Using a Web page is one way to get around the problem. A Web page can act as a front end for the administrator with all work being done on the server rather than the client. This solution is limited because absolutely everything will get downloaded every time the administrator needs to do anything with the server from a remote terminal. The biggest problems with this solution are speed and flexibility.

Writing a Utility-level Tool

The vast majority of the custom-built administrative tools that you currently have probably fall into the utility category. A utility-level tool is small and fast and usually performs one or two tasks exceptionally well. Because of its limited functionality, this kind of tool is relatively easy to learn. In many cases, the programmer building such a tool includes minimal help simply because the tool doesn't do much (this practice drives some network administrators nuts because the lack of documentation makes it unclear just what the tool does do).

This section shows you how to build a very simple utility-level tool that will allow you to compare the version requirements for an application with those of the current operating system. It's based on a new function for Windows 2000 called `VerifyVersionInfo()`. The example code returns one of three results. If the current operating system does meet the required minimum, then you see a success message. If the operating system doesn't meet the required minimum, you see an operating system error message. Finally, if the `VerifyVersionInfo()` function isn't supported by the current operating system, you see a function not supported error message. The following sections show you how to create this example.



CD-ROM Location: Chapt21\VersionInfo

Program Name: VersionInfo.EXE

Source Code Modules in Text: VersionInfoDlg.CPP, VersionInfoDlg.h, VersionInfo.RC

Creating the Program Shell

Let's begin by creating the application shell. The following procedure will get you started (it assumes that you've already completed the basic examples in Chapter 3 and uses an abbreviated procedure).

Try This

1. Open Visual C++ if you haven't done so already.
2. Use the File | New command to display a New dialog box. Click the Projects tab.
3. Highlight the MFC AppWizard (exe) icon.
4. Type a name for your program in the Project Name field. The sample program uses the name **VersionInfo**.
5. Click OK. You'll see the MFC AppWizard Step 1 dialog box.
6. Choose the Dialog based option, then click Next. You'll see the MFC AppWizard Step 2 of 4 dialog box.

7. Change the dialog box title bar text to “Check the Operating System Version Requirements.”
8. Click Finish. You’ll see the New Project Information dialog box shown in Figure 21-1.
9. Click OK. Visual C++ will generate your application. You’ll see a Dialog Editing Window.



Figure 21-1: The New Project Information dialog box shows that we’re creating a dialog-based application with standard features.

Designing an Interface

At this point, you should have a simple dialog-based application shell. Now it’s time to design the dialog box that the user will see when using the remote component. Figure 21-2 shows the design that we’ll use in this case. As you can see, the new `VerifyVersionInfo()` function allows you to check quite a few operating system parameters. Each edit box in Figure 21-2 represents one parameter that you can check. The pushbutton allows you to initiate the check once you’ve entered any required parameters. Table 21-1 will tell you how to configure the various controls for use.



Figure 21-2: The Check the Operating System Version Requirements dialog box allows you to enter the operating system parameters that you want to check.

Table 21-1
DCOM Test Component Settings

<i>Component</i>	<i>Property</i>	<i>Setting</i>
IDC_CHK_VERSION	Caption	Check Version Information
	Multiline	Checked
	Horizontal Size	60
IDOK	Horizontal Size	60
IDCANCEL	Horizontal Size	60
IDC_BUILDNUMBER	Number	Checked
	Horizontal Size	70
IDC_MAJORVERSION	Number	Checked
	Horizontal Size	70
IDC_MINORVERSION	Number	Checked
	Horizontal Size	70

Continued

Table 21-1 (continued)

Component	Property	Setting
IDC_PLATFORMID	Number	Checked
	Horizontal Size	70
IDC_SERVICEPACKMAJOR	Number	Checked
	Horizontal Size	70
IDC_SERVICEPACKMINOR	Number	Checked
	Horizontal Size	70
IDC_SUITEMASK	Number	Checked
	Horizontal Size	70
IDC_PRODUCTTYPE	Number	Checked
	Horizontal Size	70

Now that you have a dialog box with some controls in it, we need to make those controls accessible to the rest of the application. The following procedure shows you how to create the member variables required for this example.

Try This

1. Ctrl+Double-click IDC_BUILDNUMBER. You'll see the Add Member Variable dialog box shown in Figure 21-3.

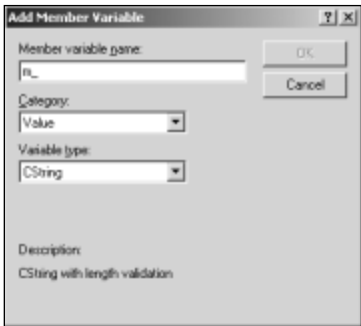


Figure 21-3: The Add Member Variable dialog box allows you to add member variables to access the dialog box components.

2. Type **m_buildNumber** in the Member variable name field.
3. Choose Control in the Category field. Visual C++ will automatically choose CEdit in the Variable type field for you.

4. Click OK. Visual C++ will add the new member variable to your application.
5. Repeat Steps 1 through 4 for the remaining edit box controls. Use **m_majorVersion**, **m_minorVersion**, **m_platformID**, **m_servicePackMajor**, **m_servicePackMinor**, **m_suiteMask**, and **m_productType** as the member variable names.
6. Use the View | Class Wizard command to display the MFC ClassWizard dialog box. Select the Member Variables tab and choose CVersionInfoDlg in the Class name field. Your member variables match those shown in Figure 21-4.
7. Close the MFC ClassWizard dialog box.

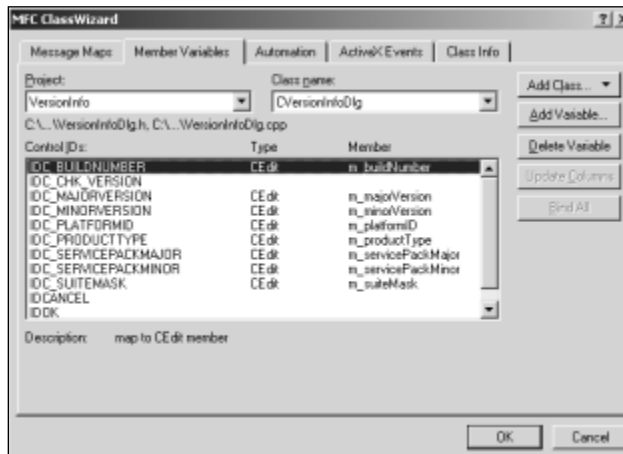


Figure 21-4: The example program requires member variables for each of the edit box components.

Adding Some Code

Our simple application is ready for some code. The first thing you need to do is to add a member function to respond to the Check Version Information pushbutton. Double-click the Check Version Information pushbutton and you'll see an Add Member Function dialog box like the one shown in Figure 21-5. Click OK to accept the default function name. Listing 21-1 shows the code you need to add to the `OnChkVersion()` function to make this application operational.



Figure 21-5: The Add Member Function dialog box allows you to add a handler for the Check Version Information pushbutton.

Listing 21-1: OnChkVersion() Source Code

```

void CVersionInfoDlg::OnChkVersion()
{
    OSVERSIONINFOEX osi;                // OS version info.
    CString          oValue;             // Input value.
    DWORDLONG        dwlConditionMask;   // Condition Mask.
    LPVOID           lpMsgBuf;           // Error Message Buffer

    // Tell GetVersionEx() that this is a Windows 2000 call.
    osi.dwOSVersionInfoSize = sizeof(OSVERSIONINFOEX);

    // Get the input values and place them in the information
    // structure.
    m_buildNumber.GetWindowText(oValue);
    osi.dwBuildNumber = atoi(oValue);
    m_majorVersion.GetWindowText(oValue);
    osi.dwMajorVersion = atoi(oValue);
    m_minorVersion.GetWindowText(oValue);
    osi.dwMinorVersion = atoi(oValue);
    m_platformID.GetWindowText(oValue);
    osi.dwPlatformId = atoi(oValue);
    m_servicePackMajor.GetWindowText(oValue);
    osi.wServicePackMajor = atoi(oValue);
    m_servicePackMinor.GetWindowText(oValue);
    osi.wServicePackMinor = atoi(oValue);
    m_suiteMask.GetWindowText(oValue);
    osi.wSuiteMask = atoi(oValue);
    m_productType.GetWindowText(oValue);
    osi.wProductType = atoi(oValue);

    // Build the condition mask. You must call this macro once
    // for each value being compared by VerifyVersionInfo().
    VER_SET_CONDITION(
        dwlConditionMask,
        VER_BUILDNUMBER,
        VER_GREATER_EQUAL);

    VER_SET_CONDITION(
        dwlConditionMask,
        VER_MAJORVERSION,
        VER_GREATER_EQUAL);

    VER_SET_CONDITION(
        dwlConditionMask,
        VER_MINORVERSION,
        VER_GREATER_EQUAL);

    VER_SET_CONDITION(
        dwlConditionMask,
        VER_PLATFORMID,
        VER_GREATER_EQUAL);
}

```

```

VER_SET_CONDITION(
    dwlConditionMask,
    VER_SERVICEPACKMAJOR,
    VER_GREATER_EQUAL);

VER_SET_CONDITION(
    dwlConditionMask,
    VER_SERVICEPACKMINOR,
    VER_GREATER_EQUAL);

VER_SET_CONDITION(
    dwlConditionMask,
    VER_SUITENAME,
    VER_AND);

VER_SET_CONDITION(
    dwlConditionMask,
    VER_PRODUCTTYPE,
    VER_GREATER_EQUAL);

// Verify that the required version information matches
// the requested version information.
if (VerifyVersionInfo(
    &osi,
    VER_BUILDNUMBER |
    VER_MAJORVERSION |
    VER_MINORVERSION |
    VER_PLATFORMID |
    VER_SERVICEPACKMAJOR |
    VER_SERVICEPACKMINOR |
    VER_PRODUCTTYPE |
    VER_SUITENAME,
    dwlConditionMask))

    // If so, then tell the user the system is ok.
    AfxMessageBox("The System is OK");

else
    // If not, see if the system supports
    // VerifyVersionInfo.
    if (GetLastError() == ERROR_OLD_WIN_VERSION)

        // If it does, then the version requirements are
        // too high.
        AfxMessageBox("Update Operating System.");

    // Otherwise, the operating system doesn't provide
    // the required support or something else went wrong.
    else
    {

```

Continued

Listing 21-1: (continued)

```

        // Format an error message base on GetLastError().
        FormatMessage(
            FORMAT_MESSAGE_ALLOCATE_BUFFER |
            FORMAT_MESSAGE_FROM_SYSTEM |
            FORMAT_MESSAGE_IGNORE_INSERTS,
            NULL,
            GetLastError(),
            MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
            (LPTSTR) &lpMsgBuf,
            0,
            NULL);

        // Display the error message.
        AfxMessageBox((LPCTSTR)lpMsgBuf);

        // Free the buffer.
        LocalFree( lpMsgBuf );
    }
}

```

There are several things going on in the `OnChkVersion()` method. The first thing we need to do is fill out the `OSVERSIONINFOEX` data structure. This data structure tells the `VerifyVersionInfo()` function what values to use for comparison purpose. The first problem we have to take care of, though, is that there are actually two data structures in use at present: `OSVERSIONINFO` and `OSVERSIONINFOEX`. Placing the correct value inside the `dwOSVersionInfoSize` member of the data structure ensures that the operating system won't get confused and use values for the wrong data structure.

Getting the remaining values from the edit boxes is relatively simple. All you need to do is get the data, place it in a string (a `CString` class variable in this case), then use `atoi()` to convert it to a number and place it in the data structure.

**Note**

There's a bug in the current version of `VerifyVersionInfo()`. You need to completely fill the `OSVERSIONINFOEX` data structure with values, even if you don't intend to check every value that the data structure contains against the operating system. Otherwise, you'll get one of several error messages, the most common of which is, "The argument string passed to `DosExecPgm` isn't correct." The only exception to this rule appears to be the `szCSDVersion` data member.

Three of the data members require special mention. The first is `dwPlatformId`. This data member can currently have one of three values: Win32s on Windows 3.1 (`VER_PLATFORM_WIN32S`), Win32 on Windows 95 or Windows 98 (`VER_PLATFORM_WIN32_WINDOWS`), or Win32 on Windows NT or Windows 2000 (`VER_PLATFORM_WIN32_NT`). Depending where you run the application, you get

one of these three values. As previously mentioned, `VerifyVersionInfo()` will run only on Windows 2000 at the moment.

The second data member that deserves special mention is `dwSuiteMask`. This is actually a bit mask that tells you whether one or more of a series of products are installed on the current machine. The values include Microsoft Small Business Server (`VER_SUITE_SMALLBUSINESS`); Windows NT Server Enterprise Edition (`VER_SUITE_ENTERPRISE`); Microsoft BackOffice (`VER_SUITE_BACKOFFICE`); Windows-based Terminal Server (`VER_SUITE_TERMINAL`); and Microsoft Small Business Server with a Restrictive License (`VER_SUITE_SMALLBUSINESS_RESTRICTED`). There's also a special value for some types of communications servers (`VER_SUITE_COMMUNICATIONS`).

Finally, the `wProductType` data member tells you which of several Windows NT (or Windows 2000) platforms are currently in use. The current values include Windows NT Workstation (`VER_NT_WORKSTATION`), Windows NT Domain Controller (`VER_NT_DOMAIN_CONTROLLER`), and Windows NT Server (`VER_NT_SERVER`). Microsoft will likely add values to this list to take special Windows 2000 configurations into account by the time you read this. However, these were the only values in use as of the time of writing.

We also have to build a condition mask for each comparison that we want to perform. The first thing to notice is that `dwlConditionMask` is of type `DWORDLONG`, which is a 64-bit value. The second thing to notice is that we don't reinitialize `dwlConditionMask` after each call to the `VER_SET_CONDITION()` macro. The macro calls are cumulative. You have to call the macro once for each comparison that you want to perform. Because we have eight comparisons in the example, we need to call `VER_SET_CONDITION()` eight times—once with each constant for the various values we want to check. Note that there are a variety of constants that you can use for comparison purposes, including `VER_EQUAL`; `VER_GREATER`; `VER_GREATER_EQUAL`; `VER_LESS`; and `VER_LESS_EQUAL`. Only the `wSuiteMask` (`VER_SUITENAME` constant) data member requires special handling. You'll use `VER_AND` (if all of the suites are required to meet the minimum requirements) or `VER_OR` (if any one of them will satisfy the minimum requirements) with the `wSuiteMask` data member.

Once you have the `OSVERSIONINFOEX` data structure and the condition mask built, you can use `VerifyVersionInfo()` to make the comparison. Notice that we still need to include constants for each of the version information statistics we want to compare as part of the function call. Make absolutely certain that your condition mask and the list of comparison constants you provide match. Otherwise, you'll get an error message back that won't really tell you much with regard to what went wrong with the function call (normally, you'll get some type of bad value message).

There are three outcomes of the `VerifyVersionInfo()` call, as shown in the source code. If the call is successful, then the operating system meets the minimum requirements necessary to install the application. On the other hand, if you get an

ERROR_OLD_WIN_VERSION error message back, then the operating system won't support the application. There are actually two other possibilities, but they're both taken care of by one return type. If you didn't provide the right values, or if the operating system doesn't support `VerifyVersionInfo()`, you'll get an error message other than ERROR_OLD_WIN_VERSION back. Because there isn't any way to know what that error message will be, it's best to use the `FormatMessage()` function as shown in the code to create a human-readable text form of the error result. Once you get the string back, you can display it on screen. Make sure you free the message buffer as shown in the code or you'll end up with a memory leak.

In addition to the code required to check the version information required for an application against the operating system version information, it might be nice to display the current operating system information on startup. You can do that using the `GetVersionEx()` function. Add this code to the `OnInitDialog()` method and you'll see that version information on startup. Listing 21-2 shows the code you need to add.

Listing 21-2: `OnInitDialog()` Method Additions

```
// Create a data structure to hold the current
// operating system version information and a CString to
// hold the converted values.
OSVERSIONINFOEX    osi;
CString             oValue;

// Tell GetVersionEx() that this is a Windows 2000 call.
osi.dwOSVersionInfoSize = sizeof(OSVERSIONINFOEX);

// Get the version information.
GetVersionEx((OSVERSIONINFO*)&osi);

// Fill the edit boxes with values.
itoa(osi.dwBuildNumber, oValue.GetBuffer(40), 10);
oValue.ReleaseBuffer(-1);
m_buildNumber.SetWindowText(oValue);
itoa(osi.dwMajorVersion, oValue.GetBuffer(40), 10);
oValue.ReleaseBuffer(-1);
m_majorVersion.SetWindowText(oValue);
itoa(osi.dwMinorVersion, oValue.GetBuffer(40), 10);
oValue.ReleaseBuffer(-1);
m_minorVersion.SetWindowText(oValue);
itoa(osi.dwPlatformId, oValue.GetBuffer(40), 10);
oValue.ReleaseBuffer(-1);
m_platformID.SetWindowText(oValue);
itoa(osi.wServicePackMajor, oValue.GetBuffer(40), 10);
oValue.ReleaseBuffer(-1);
m_servicePackMajor.SetWindowText(oValue);
itoa(osi.wServicePackMinor, oValue.GetBuffer(40), 10);
oValue.ReleaseBuffer(-1);
m_servicePackMinor.SetWindowText(oValue);
```

```
itoa(osi.wSuiteMask, oValue.GetBuffer(40), 10);
oValue.ReleaseBuffer(-1);
m_suiteMask.SetWindowText(oValue);
itoa(osi.wProductType, oValue.GetBuffer(40), 10);
oValue.ReleaseBuffer(-1);
m_productType.SetWindowText(oValue);
```

This code isn't all that different from the code we just discussed. Obviously, we're asking `GetVersionEx()` to fill the `OSVERSIONINFOEX` data structure. Notice that we have to typecast the data structure as an `OSVERSIONINFO` data structure. Again, it's very important to specify the data structure size in the `dwOSVersionInfoSize` data member so that the `GetVersionEx()` function knows what kind of data structure it should fill.

Once the data structure is full, a `CString` is used to grab the various values. You can use the `itoa()` function to convert the numeric values provided in the `OSVERSIONINFOEX` data members to strings. Make sure you release the `CString` buffer before you attempt to use the associated variable again.

Testing the Results

At this point, you have a utility that's ready to compile and use. The whole purpose of this utility is to see how the `GetVersionEx()` and `VerifyVersionInfo()` functions work. Of course, you can also add this code directly to your own applications. When used within a setup program, it'll help you decide whether the user's operating system requires update or not. Figure 21-6 shows the `VersionInfo` utility in operation.

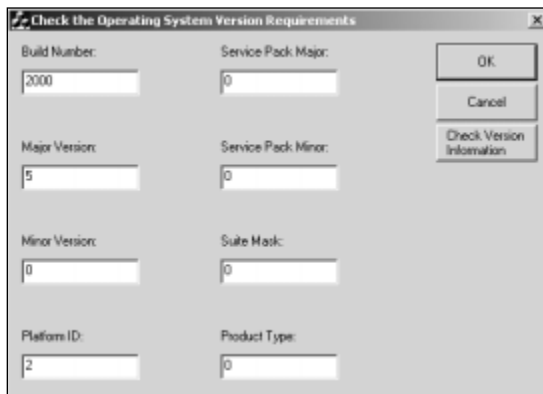


Figure 21-6: The `VersionInfo` utility allows you to test how the `VerifyVersionInfo()` function will work under varying conditions.

Writing an MMC Snap-In

Windows 2000 comes with a fully implemented version of Microsoft Management Console (MMC). This utility allows you to add snap-ins to adjust its current feature set and to add functionality. In essence, MMC allows you to use one interface to perform any number of administrative tasks.

This section shows you how to build a very simple MMC snap-in. This snap-in uses two new Windows 2000 functions to display and change the names of the various network servers that you have installed. The `GetComputerNameEx()` function allows you to display the computer's NetBIOS and DNS names in various formats (we explore them all). Likewise, the `SetComputerNameEx()` function allows you to change these names. The sections that follow get you started creating and testing an MMC snap-in.



CD-ROM Location: Chapt21\ComputerName

Program Name: ComputerName.DLL

Source Code Modules in Text: CompName.CPP, CompName.h, CompName.RC

An Overview of the Interfaces

Like everything else in the world of COM, writing an MMC snap-in requires at least a passing acquaintance with the many interfaces that you can use. Unless you know which interfaces are available and have some idea of what they're used for, you can't really communicate with MMC very well. Obviously, the first part of the equation is MMC. It's important to know what you can expect in the way of services for your component. The following list provides a very brief outline of the various interfaces that MMC supports.

IConsole2

This is the main interface that allows your component to communicate with the console. It allows you to do things such as set the Result View Pane column heading text. You'll get a pointer to this interface through the `IComponent::Initialize()` and `IComponentData::Initialize()` methods.

IConsoleNamespace2

This interface allows you to enumerate subcontainers within the Scope Pane. For example, you might create a main node for all of the employees in your company and a series of subcontainers that contain the employee name and other pertinent information.

<code>IConsoleVerb</code>	Think of action words when you think about this interface. It allows you to request actions from the console such as cut, paste, copy, delete, properties, rename, refresh, and print.
<code>IContextMenuCallback</code>	This interface allows you to add menu items to a context menu. There's one default context menu associated with the Scope Pane and the Result View Pane for your snap-in within MMC. In most cases, the default context menu for a snap-in contains a single entry, Help. You need to implement any additional context menu items separately.
<code>IContextMenuProvider</code>	Allows you to create custom context menus for your snap-in. You'd use this when the default context menu would require too many changes to implement property for a particular item in either MMC pane. For example, you might write a wrapper for an ActiveX control that requires a different context menu than the main snap-in does.
<code>IControlbar</code>	Use this interface to create control bars for your snap-in. These control bars work just like any application control bar—the main difference is that you're implementing them within MMC instead of a standard application frame.
<code>IDisplayHelp</code>	Not every snap-in requires extensive help, but you should at least provide instructions on how to use the snap-in and what the various elements are used for. This interface allows you to add the custom help for your snap-in to the main MMC help. As far as the user is concerned, the help you add was part of the main MMC help file all the time.
<code>IHeaderCtrl</code>	This interface allows you to modify and manipulate the Result View Pane column headings. You can add or remove columns, change the text they contain, and change their width using the methods in this interface.
<code>IToolbar</code>	Use this interface to create toolbars for your snap-in. These toolbars work just like any application toolbar—the main difference is that you're implementing them within MMC instead of a standard application frame.

<code>IImageList</code>	This is one of the more common interfaces that you'll implement because it contains the list of images that you'll use to display items in both the Scope Pane and Result View Pane. Normally, you'll need a minimum of two icons: a 16 × 16-pixel image for small icon item displays and a 32 × 32-pixel image for large icon item displays. In addition, you may want to add an icon for each of the node types you add to the MMC display. The image list uses virtual index numbers because it gets shared by all the snap-ins currently loaded by MMC. The index number you provide for an image gets mapped to the real number within the image list automatically.
<code>IMenuButton</code>	This interface allows you to add, delete, or modify custom buttons on the menu bar. These buttons are associated with your snap-in and allow the user to perform special tasks within your snap-in without having to rely on actual menu entries.
<code>IPropertySheetCallback</code>	Use this interface to add or remove a property page to the property sheet for your snap-in.
<code>IPropertySheetProvider</code>	This is the interface that you use to create, find, or show a property sheet for your snap-in. Extension snap-ins can also use this interface to add extension property sheets to the property sheet created by the main snap-in.
<code>IResultData</code>	You always need to use this interface because it contains the methods required to add items to the Result View Pane. This interface also contains all of the required item management methods, allows you to define the total number of items displayed in the Result View Pane, and perform visual modifications like sorting the items currently displayed in the Result View Pane.
<code>IRequiredExtensions</code>	Use this interface to add and enable any extension snap-ins required by your snap-in.

COM communication is definitely a two-way street. You have to implement certain interfaces within your snap-in to make it work at all. In addition, there are optional interfaces that you can implement to extend the functionality of your snap-in. In fact, as we've seen in other chapters, the interfaces that you implement often determine the functionality of your component and give it a unique personality

when compared to other components of nearly the same type. The following list will provide you with an overview of the required and optional interfaces for an MMC snap-in (this list isn't inclusive—it doesn't include common interfaces that generic components might implement).

IComponent	This is one of several required interfaces that allow MMC to communicate with your component. You override the methods in this interface to provide a unique implementation of the Result View Pane. The two most commonly overridden methods are <code>Initialize()</code> and <code>GetDisplayInfo()</code> . We look at these methods later in the chapter. This interface corresponds to the “view” level of Microsoft’s document/view architecture. Most snap-ins use this interface to work with the Result View Pane of MMC.
IComponentData	This is the second of several required interfaces that allow MMC to communication with your component. In this case, you override the default methods to allow MMC to work with the data provided by your component. This interface corresponds to the “document” level of Microsoft’s document/view architecture. Most snap-ins use this interface to work with the Scope Pane of MMC.
IExtendContextMenu	This interface allows a snap-in extension to add new entries to a context menu.
IExtendControlbar	This interface allows a snap-in extension to add a custom control bar to MMC.
IExtendPropertySheet2	This interface allows a snap-in extension to add property pages to the property sheet for an item. You can also use the methods in this interface to query existing pages and to obtain the watermark and header information for existing property pages.
IEnumTASK	Use this interface with the <code>IExtendTaskPad</code> interface. It allows you to enumerate the tasks that will be added to a taskpad.
IResultDataCompare	This interface allows your snap-in to compare items in the Result View Pane against one another when they’re in sorted order.

ISnapinAbout	Even though this interface wouldn't be required from a strict functionality perspective, most snap-ins implement it. The methods in this interface allow the snap-in to display information about itself so that the user can learn more about the snap-in without too much difficulty. Most of the information obtained using this interface is simple. For example, MMC can call on the <code>GetProvider()</code> method to learn who created the snap-in.
IResultOwnerData	This is an optional interface that allows you to create virtual lists. Although it does take some time to implement this interface, you'll notice a performance increase in some types of data-intensive snap-ins by doing so.
ISnapinHelp	This interface allows MMC to determine the location of the snap-ins help file. The file is then added to the main MMC help file so that the user can more easily locate information about the snap-in while looking through the standard help file.
IToolbar	Use this interface to create, delete, and manipulate toolbars associated with your snap-in. Each toolbar is created in its own band on the control bar.

Creating the Snap-In Shell

This section shows how to create the ATL shell for the MMC component that we build for this example. Once we have a shell in place, we can add an MMC component to it. The procedure in this section is a very simplified version of the procedure in Chapter 13, where we created the first ATL-based component in the book. Make sure you're familiar with the basics of creating a component using ATL before you create the example in this section.

Try This

1. Start Visual C++, if you haven't done so already.
2. Use the File | New command to display the New dialog box. Click the Projects tab.
3. Highlight the ATL COM AppWizard icon.
4. Type a project name in the Project name field, such as **ComputerName**, which is used in the example.

5. Click OK. You'll see the ATL COM AppWizard – Step 1 of 1 dialog box.
6. Choose the Dynamic Link Library (DLL) option. Remember, we're creating an in-process server that will execute as an extension of the MMC framework.

Tip

This is one time when you might consider mixing MFC and ATL because an MMC component won't be used outside the machine it's loaded on. You don't have to worry about download size, and the number of compatibility problem scenarios should be reduced.

7. Check Support MFC. Although this will make the component larger and harder to distribute, it will also reduce the amount of work you need to do to create the component in the first place.
8. Click Finish. You'll see a New Project Information dialog box like the one shown in Figure 21-7.

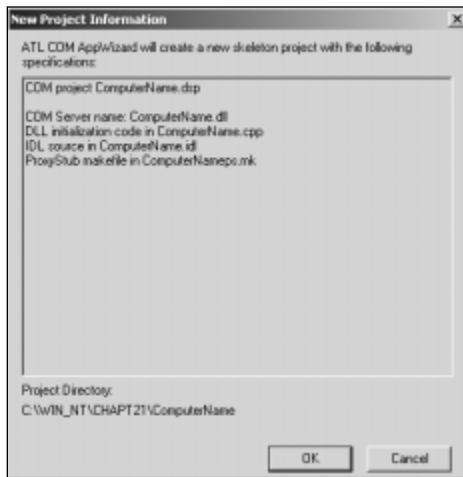


Figure 21-7: The New Project Information dialog box shows that this will be a DLL-based ATL component.

9. Click OK. Visual C++ will create the program shell for you. Remember that this shell is essentially blank; there aren't any objects included in it. The next step is to add an object.
10. Right-click the ComputerName Classes folder in the ClassView tab of the Workspace window and select New ATL Object from the context menu. You'll see an ATL Object Wizard dialog box.
11. Select the Objects option in the Category field. You'll see a list of simple objects similar to those shown in Figure 21-8.

12. Highlight the MMC SnapIn icon, then click Next. You'll see the ATL Object Wizard Properties dialog box shown in Figure 21-9.

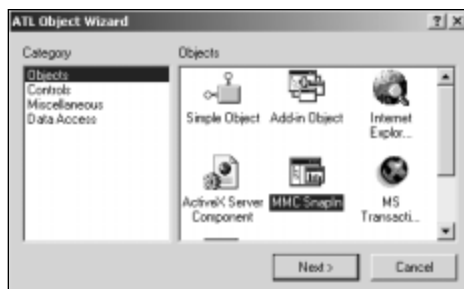


Figure 21-8: These objects aren't designed to provide a user interface unless you add the required support.

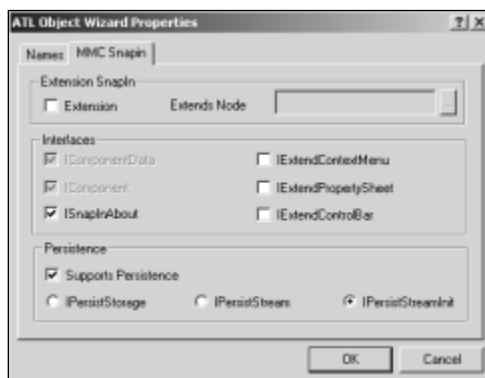


Figure 21-9: The Names tab of the ATL Object Wizard Properties dialog box allows you to assign a name for your MMC snap-in.

13. Type **CompName** in the Short Name field. The ATL Object Wizard will automatically add appropriate entries to the remaining fields.
14. Click the MMC Snapin tab. You'll see a dialog box like the one shown in Figure 21-10. This is where you define additional features for the MMC snap-in. Like most COM objects, a MMC snap-in depends on interfaces to define its functionality. In this case, there are special interfaces that you can implement to augment the basic snap-in functionality. You can also make a snap-in an extension of an existing node (another snap-in). A snap-in will default to providing persistent storage and an About dialog box. In this case, we have no need for persistent storage because the result we create is immediate.
15. Uncheck Supports Persistence. This will slightly reduce the complexity of creating the snap-in.
16. Click OK. Visual C++ will create the snap-in.

Adding Some Code

This example shows how to create a simple single node snap-in for MMC that displays various `GetComputerNameEx()` outputs. As a result, we won't be doing anything with the MMC Scope Pane.

There are two main sections of code for this example. The first processes event requests from MMC. The second displays the requested data. In most circumstances, you could combine these two tasks into one method call. However, MMC uses a callback mechanism to display items within the Result View Pane. Each item is processed on a column-by-column basis. In other words, the callback function will need to detect the current column and provide only the string required for that column, rather than fill in all of the columns at one time. Listing 21-3 contains the source code that you need to add to the `MMCN_SHOW` event handler contained in the `Notify()` method in bold type.

Listing 21-3: **MMCN_SHOW** Code Additions

```
switch (event)
{
case MMCN_SHOW:
    {
        RESULTDATAITEM    Item;    // Result data variable.

        CComQIPtr<IResultData,
            &IID_IResultData> spResultData(spConsole);

        // Create two headers.
        spHeader->InsertColumn(0, L"Name Type", 0, 250);
        spHeader->InsertColumn(1, L"Name Value", 0, 150);

        // Initialize the result data variable.
        memset(&Item, 0, sizeof(RESULTDATAITEM));

        // Create entries required for first item. Include
        // constants for relevant items in mask. Make sure
        // you use a callback for strings. Set the image to
        // the 16 X 16 pixel image.
        Item.mask = RDI_STR | RDI_IMAGE | RDI_PARAM;
        Item.str = MMC_CALLBACK;
        Item.nImage = 0;
        Item.lParam = 0;
        Item.nCol = 0;

        // Display the first item.
        spResultData->InsertItem(&Item);

        // Modify lParam member for second query.
        Item.lParam = 1;
```

Continued

Listing 21-3: (continued)

```

        // Display the second item.
        spResultData->InsertItem(&Item);

        // Perform the same two steps for subsequent items.
        Item.lParam = 2;
        spResultData->InsertItem(&Item);
        Item.lParam = 3;
        spResultData->InsertItem(&Item);
        Item.lParam = 4;
        spResultData->InsertItem(&Item);
        Item.lParam = 5;
        spResultData->InsertItem(&Item);
        Item.lParam = 6;
        spResultData->InsertItem(&Item);
        Item.lParam = 7;
        spResultData->InsertItem(&Item);

        hr = S_OK;
        break;
    }
    case MMCN_EXPAND:

```

As you can see, the task of responding to MMCs request to show the component's data is relatively straightforward. The code begins by adding two headers to the MMC display. It doesn't matter what view MMC is currently using, you definitely want the headers available if the user decided to use the Detail view, rather than one of the other views, such as Large Icon. In fact, the Detail view is the only useful view for this component because the other views hide the data, and we haven't implemented a dialog box that could be used to display the data instead.

The next step, after the two column headings are displayed, is to initialize the `RESULTDATAITEM` data structure variable `Item`. This is accomplished in two steps. First, the `memset()` function sets aside the memory required for the data structure. Then, the various data structure elements get filled in. There's a special data structure element named `mask` that you must fill in with constants that indicate which of the other data structure elements contain information. In this case, we use the `str`, `nImage`, and `lParam` data members to hold actual information. You also want to initialize `nCol` to 0 to ensure the component starts displaying information at the right column.

The `str` input value is `MMC_CALLBACK`. It would appear from the Microsoft documentation that it's possible to place a string in this data member if the output consisted of a single column of data, rather than rely on a callback function. However, it's usually safer and more flexible to rely on a callback function rather than supply the `str` input value directly. You must use a callback function if you plan on displaying more than one column of data, which is likely in most situations.

Adding the data to the MMC display comes next. You'll use the `InsertItem()` method to add new data items to the display. This method requires a single argument as input, a pointer to the item that you want to add, which must be a `RESULTDATAITEM` data structure. There are also methods for removing and modifying existing items should you wish to do so.

The rest of the code consists of two steps. First, it places a new value in `lParam` to indicate which kind of data to display. Second, it uses `InsertItem()` to add the item to the display.

The ATL COM AppWizard doesn't automatically provide a callback function to use for displaying the various items on screen, so you need to add it. However, this isn't as easy as you might think because you can't add the callback function `GetDisplayInfo()` just anywhere in the code. You must add it to the class that implements either the `IComponent` or the `IComponentImpl` interface. If you use the ATL COM AppWizard that's provided with Visual C++ 6, then you'll find this interface in the `C<Component Name>Component` class. However, this interface could appear anywhere (as witnessed by the Microsoft examples provided with products such as MSDN). Because there isn't any consistent place for this interface to appear, you need to check your source code before adding the `GetDisplayInfo()` function.

Once you do find the class that implements the `IComponent` or `IComponentImpl` interface, right-click the class folder and choose **Add Member Function** from the context menu. You'll see an **Add Member Function** dialog box like the one shown in Figure 21-10. Type **STDMETHODIMP** in the **Function Type** field and **GetDisplayInfo(RESULTDATAITEM *pResultItem)** in the **Function Declaration** field. Click **OK** and Visual C++ will add the function to the class for you.

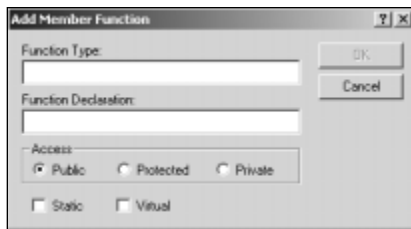


Figure 21-10: The **Add Member Function** dialog box allows you to add the `GetDisplayInfo()` method to the `CCompNameComponent` class.

At this point, you're ready to see the callback function that displays the various items on screen. This code appears in Listing 21-4. Though this code may appear long, it actually contains a few processing steps that get repeated using different values depending on the item that the caller wants to display. Because I want to show you everything that the new `GetComputerNameEx()` function is capable of doing, we need to call the function in eight different ways.

Listing 21-4: GetDisplayInfo() Source Code

```

STDMETHODIMP CCompNameComponent::GetDisplayInfo(RESULTDATAITEM *pResultItem)
{
    CString oComputerName; // Buffer to hold computer name data.
    ULONG   ulBufferSize = MAX_COMPUTERNAME_LENGTH + 1;

    // If the caller sent some information.
    if (pResultItem)

        // If that information contains a string.
        if (pResultItem->mask & RDI_STR)
        {

            // Check which string the caller is requesting.
            switch (pResultItem->lParam)
            {
            case 0:
                // Display the NetBIOS name item.
                if (pResultItem->nCol == 0)
                    pResultItem->str = (LPOLESTR)L"NetBIOS Name";
                else
                {

                    // See if there is a NetBIOS name for this item.
                    if (GetComputerNameEx(
                        ComputerNameNetBIOS,
                        oComputerName.GetBuffer(MAX_COMPUTERNAME_LENGTH + 1),
                        &ulBufferSize))
                    {

                        // Release the string buffer.
                        oComputerName.ReleaseBuffer(-1);

                        // Convert buffer contents to an OLE string.
                        USES_CONVERSION;
                        pResultItem->str = T2OLE(oComputerName.GetBuffer(40));
                    }

                    // If not, display a failure string.
                    else
                        pResultItem->str = (LPOLESTR)L"Value Not Available";
                }
            }
            break;

        case 1:

            // Display the DNS Host name item.
            if (pResultItem->nCol == 0)
                pResultItem->str = (LPOLESTR)L"DNS Host Name";
            else
            {

```

```

        // See if there is a DNS Host name for this item.
        if (GetComputerNameEx(
            ComputerNameDnsHostname,
            oComputerName.GetBuffer(MAX_COMPUTERNAME_LENGTH + 1),
            &ulBufferSize))
        {
            // Release the string buffer.
            oComputerName.ReleaseBuffer(-1);

            // Convert buffer contents to an OLE string.
            USES_CONVERSION;
            pResultItem->str = T2OLE(oComputerName.GetBuffer(40));
        }

        // If not, display a failure string.
        else
            pResultItem->str = (LPOLESTR)L"Value Not Available";
    }
    break;

case 2:

    // Display the DNS Domain item.
    if (pResultItem->nCol == 0)
        pResultItem->str = (LPOLESTR)L"DNS Domain";
    else
    {
        // See if there is a DNS Domain for this item.
        if (GetComputerNameEx(
            ComputerNameDnsDomain,
            oComputerName.GetBuffer(MAX_COMPUTERNAME_LENGTH + 1),
            &ulBufferSize))
        {
            // Release the string buffer.
            oComputerName.ReleaseBuffer(-1);

            // Convert buffer contents to an OLE string.
            USES_CONVERSION;
            pResultItem->str = T2OLE(oComputerName.GetBuffer(40));
        }

        // If not, display a failure string.
        else
            pResultItem->str = (LPOLESTR)L"Value Not Available";
    }
    break;

case 3:

```

Continued

Listing 21-3: *(continued)*

```

        // Display the Fully Qualified DNS Host name item.
        if (pResultItem->nCol == 0)
            pResultItem->str = (LPOLESTR)L"DNS Host Name (Fully
Qualified)";
        else
        {
            // See if there is a Fully Qualified DNS Host name for this
            item.
            if (GetComputerNameEx(
                ComputerNameDnsFullyQualified,
                oComputerName.GetBuffer(MAX_COMPUTERNAME_LENGTH + 1),
                &ulBufferSize))
            {
                // Release the string buffer.
                oComputerName.ReleaseBuffer(-1);

                // Convert buffer contents to an OLE string.
                USES_CONVERSION;
                pResultItem->str = T2OLE(oComputerName.GetBuffer(40));
            }

            // If not, display a failure string.
            else
                pResultItem->str = (LPOLESTR)L"Value Not Available";
        }
        break;

    case 4:

        // Display the Physical NetBIOS name item.
        if (pResultItem->nCol == 0)
            pResultItem->str = (LPOLESTR)L"NetBIOS Name (Physical)";
        else
        {
            // See if there is a Physical NetBIOS name for this item.
            if (GetComputerNameEx(
                ComputerNamePhysicalNetBIOS,
                oComputerName.GetBuffer(MAX_COMPUTERNAME_LENGTH + 1),
                &ulBufferSize))
            {
                // Release the string buffer.
                oComputerName.ReleaseBuffer(-1);

                // Convert buffer contents to an OLE string.
                USES_CONVERSION;
                pResultItem->str = T2OLE(oComputerName.GetBuffer(40));
            }
        }
    }
}

```

```

        // If not, display a failure string.
        else
            pResultItem->str = (LPOLESTR)L"Value Not Available";
    }
    break;
case 5:

    // Display the Physical DNS Host name item.
    if (pResultItem->nCol == 0)
        pResultItem->str = (LPOLESTR)L"DNS Host Name (Physical)";
    else
    {

        // See if there is a Physical DNS Host name for this item.
        if (GetComputerNameEx(
            ComputerNamePhysicalDnsHostname,
            oComputerName.GetBuffer(MAX_COMPUTERNAME_LENGTH + 1),
            &ulBufferSize))
        {

            // Release the string buffer.
            oComputerName.ReleaseBuffer(-1);

            // Convert buffer contents to an OLE string.
            USES_CONVERSION;
            pResultItem->str = T2OLE(oComputerName.GetBuffer(40));
        }

        // If not, display a failure string.
        else
            pResultItem->str = (LPOLESTR)L"Value Not Available";
    }
    break;
case 6:

    // Display the Physical DNS Domain name item.
    if (pResultItem->nCol == 0)
        pResultItem->str = (LPOLESTR)L"DNS Domain (Physical)";
    else
    {

        // See if there is a Physical DNS Domain name for this item.
        if (GetComputerNameEx(
            ComputerNamePhysicalDnsDomain,
            oComputerName.GetBuffer(MAX_COMPUTERNAME_LENGTH + 1),
            &ulBufferSize))
        {

            // Release the string buffer.
            oComputerName.ReleaseBuffer(-1);
            // Convert buffer contents to an OLE string.
            USES_CONVERSION;
            pResultItem->str = T2OLE(oComputerName.GetBuffer(40));
        }
    }
}

```

Continued

Listing 21-3: (continued)

```

        // If not, display a failure string.
    else
        pResultItem->str = (LPOLESTR)L"Value Not Available";
    }
    break;

case 7:

    // Display the Physical Fully Qualified DNS Host name item.
    if (pResultItem->nCol == 0)
        pResultItem->str = (LPOLESTR)L"DNS Host Name (Fully Qualified
& Physical)";
    else
    {
        // See if there is a Physical Fully Qualified DNS Host name
for this item.
        if (GetComputerNameEx(
            ComputerNamePhysicalDnsFullyQualified,
            oComputerName.GetBuffer(MAX_COMPUTERNAME_LENGTH + 1),
            &ulBufferSize))
        {
            // Release the string buffer.
            oComputerName.ReleaseBuffer(-1);

            // Convert buffer contents to an OLE string.
            USES_CONVERSION;
            pResultItem->str = T2OLE(oComputerName.GetBuffer(40));
        }

        // If not, display a failure string.
    else
        pResultItem->str = (LPOLESTR)L"Value Not Available";
    }
    break;
}

}

return S_OK;
}

```

This example requires two variables, one of which is initialized to `MAX_COMPUTERNAME_LENGTH + 1`. You need to initialize the buffer length to the maximum computer name string size and then add 1 for a NULL termination to ensure that you can display the entire computer name. In addition, the Microsoft

documentation warns that using a shorter string can have unexpected results under Windows 95.

The next task is to check whether the caller has provided a filled (or at least initialized) `RESULTDATAITEM` data structure. The code also verifies that the data structure contains an initialized `str` data element. The reason is simple: This component provides text output, so the `str` data element is absolutely essential. Figuring out what to display comes next; as previously stated, there are eight different computer name values that you can obtain using the `GetComputerNameEx()` function.

The processing sequence for all eight outputs is the same, so I describe only one. There are two columns for data output. The first column contains a string that shows what type of computer name data we're displaying. Because this value is constant, we can simply place it in the `str` data element for immediate display. Notice that the code outputs an OLE string, not a standard C string. Make sure that you perform any required data conversions before you place a value in the data structure, or you'll get memory read errors when using MMC.

The second column contains the actual value for the computer name type that we want to display. This means using the `GetComputerNameEx()` function, which requires three inputs: a constant containing the type of name that we want to retrieve, a pointer to a buffer that can accept the name, and size of the buffer provided. Once the `GetComputerNameEx()` function returns, you need to release the `CString` variable buffer that we provided as part of the `GetComputerNameEx()` input. Finally, the code converts the `CString` variable contents to an OLE string for output. Make sure you add the `USES_CONVERSION` macro if you use the `T2OLE()` function shown in Listing 21-4 to perform the conversion.

Obviously, not every attempt to get a computer name is successful. In most cases, there probably isn't a name to get. With this in mind, you also need to provide a means for displaying an alternative string for the user. In this case, the component displays a simple "Value Not Available" string.

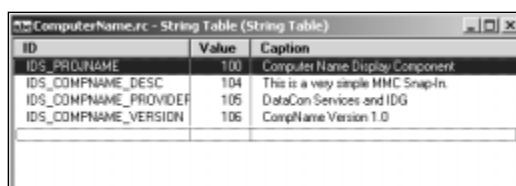
Modifying the Resource Strings

Unlike many application programming projects where customizing the various resource strings isn't a very big deal, you definitely want to perform this task when working on an MMC snap-in. The reason is simple: The resource string contents get displayed when the administrator adds the snap-in to MMC. In other words, instead of being hidden, these resource strings are the very first thing that the administrator sees.

Two strings in particular are important: `IDS_COMPNAME_DESC` and `IDS_COMPNAME_PROVIDER`. The first provides a description of the component so

that the administrator has some idea of what the component will do when installed. The second provides the name of the individual or company that created the component.

Modifying these strings is easy. Just click the ResourceView tab in the Workspace window, open the String Table folder, and then double-click the String Table entry. You'll see a window like the one shown in Figure 21-11. At this point, you can double-click the entry that you want to change, type a new value, and then click the Close button. Figure 21-11 shows the suggested string values for this example.



ID	Value	Caption
IDS_PROJNAME	100	Computer Name Display Component
IDS_COMFNAME_DESC	104	This is a very simple MMC Snap-In
IDS_COMFNAME_PROVIDER	105	DataCon Services and IDG
IDS_COMFNAME_VERSION	106	CompName Version 1.0

Figure 21-11: Modifying the String Table entries is very important when working with an MMC snap-in component.

Testing the Snap-In within MMC

At this point, you can compile the MMC snap-in and place it on the target server. Use RegSvr32 to register the component so that MMC sees it. (Just type **RegSvr32 ComputerName** at the command prompt and press Enter.) Make sure you get a registration success message, like the one shown in Figure 21-12, after you register the component. It's usually a good idea to place all of your test components in a special directory so that you can find them easier later and unregister them.

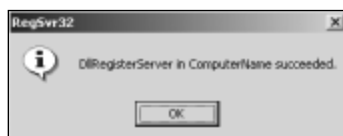


Figure 21-12: Make sure you get a successful installation of your MMC snap-in before testing it.

It's time to see what the component looks like in action. Windows 2000 automatically installs a utility named Windows 2000 Configure Your Server. In general, this application starts automatically when you start Windows 2000. You can also start it using the Start Menu | Programs | Administrative Tools | Configure Your Server command. Although you could test the ComputerName component from any of several places, one of the better places to start is at the Component Services section of the Windows 2000 Configure Your Server dialog box shown in Figure 21-13. The following procedure helps you test the ComputerName component now that you have access to the Windows 2000 administrative tools.



Figure 21-13: The Windows 2000 Configure Your Server utility provides access to most of the administrative tools that Windows 2000 has to offer.

Try This

1. Click the Manage link and you'll see a Component Services dialog box. Most of the tools used to manage Windows 2000 rely on MMC, so don't be surprised if the user interface looks the same from tool to tool.
2. Use the Console | Add/Remove Snap-In command to display the Add/Remove Snap-In dialog box.
3. Click Add. You'll see an Add Standalone Snap-In dialog box like the one shown in Figure 21-14. The figure shows the CompName component highlighted. To the right of this component is the provider string (IDS_COMPNAME_PROVIDER) that we typed into the Sting Table. Likewise, you see the description (IDS_COMPNAME_DESC) that we provided in the Description field of the dialog box.

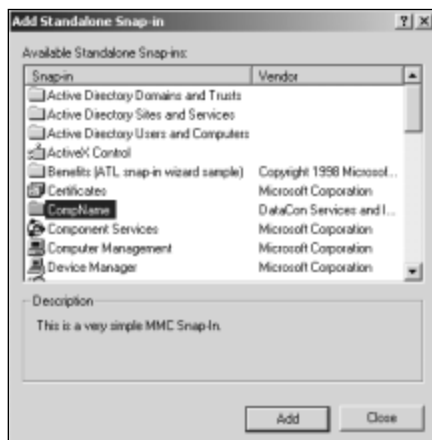


Figure 21-14: The Add Standalone Snap-In dialog box shows the importance of filling out the String Table resource.

4. Click Add and then Close. You'll see the new snap-in added to the Add/Remove Snap-in dialog box as shown in Figure 21-15.

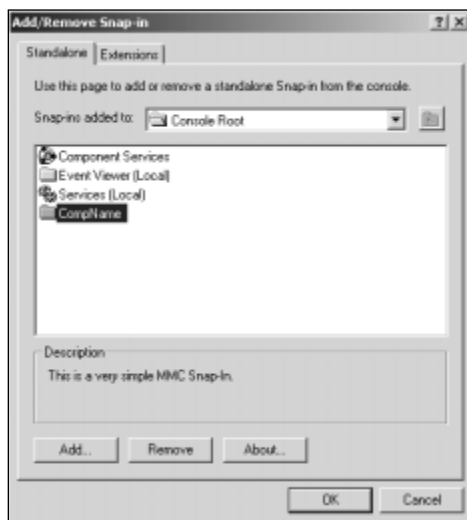


Figure 21-15: The Add/Remove Snap-in dialog box shows which components are currently installed for MMC.

5. Click OK. You'll see the CompName component added to the Component Services display. Figure 21-16 shows an example of this component in action.

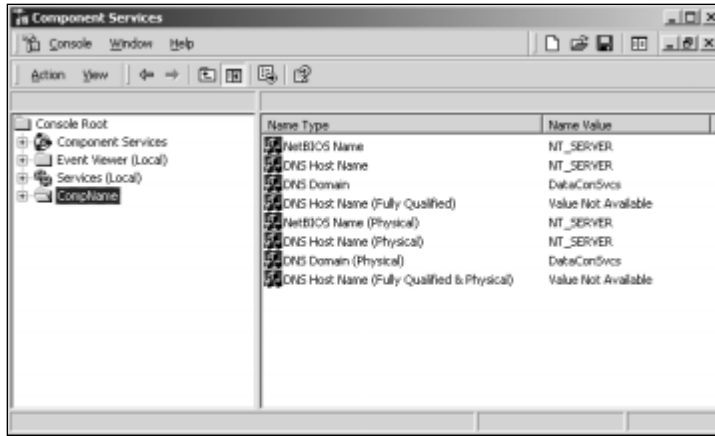


Figure 21-16: The Component Services display is one good place to see your MMC snap-in in action.

6. Use the Console | Exit command to exit MMC. MMC will ask if you want to save the current settings.
7. Click No. The Component Services dialog box will close.

Writing an HTML-accessible Tool

Browsers are beginning to form the basis of a lot of tools, not just tools for administrators. The reason is simple: A browser displays a lot of different types of information, provides a method for exchanging information, and requires little work on the part of the developer when it comes to design an interface. Of course, the biggest reason that browsers are gaining in popularity is that you can use them in a wide variety of locations both local and remote. (A browser still isn't the perfect tool because there are limits to what you can do with its interface, but the latest browsers are a big improvement over the browsers originally used to cruise the Internet.)

We already looked at all of the pieces you need to create an HTML-accessible tool. In Chapter 14, we looked at the requirements for writing a DCOM application. You can use those techniques for more than just application-to-server communication; they also are handy for a wide variety of component-level applications. For example, instead of creating a standard EXE in Chapter 14, we could have created a client-side ATL component. This component could have been placed within an application, a Web page, or any other location where the user could easily access components.

The point is this: All you really need to build an administrator tool that relies on HTML instead of standard applications is a component and some type of Web page. Using an ASP script would provide you with the ultimate level of flexibility that scripts provide and the security of using a component. In sum, creating useful HTML tools for the administrator isn't any more complex than what you've done in the past for the user. The only real difference is the mindset required to build such an application.

Of course, there are some differences that you do have to consider that have little to do with the application itself. The most important difference is the amount of security required to ensure secure communications. Obviously, if you can access the network from a remote location, someone with enough tenacity to break any security measures that you have in place can also gain access to the network.

Although it's important to ensure that you have proper login security and that the data you transmit is encrypted, make sure you take a proactive approach to managing the network as well. For one thing, assume that someone will eventually break into the network. It definitely helps if you keep good logs of network activity in secure locations and create administrative tools that help detect break-ins. Never assume that remote access is anything but a nonsecure method of working with the network. Build the tools that you want to use from a remote location with this limitation in mind.

Summary

Administrative tools can take a lot of different forms and enhance the administrator's ability to perform a variety of tasks. This chapter has shown you that the tools an administrator uses are often designed more to meet a specific need than to create a new capability. In addition, the form that the tool takes is often dictated by how and where the tool is used rather than by an administrator's preference. Here are some additional conclusions that can be drawn from the contents of this chapter:

- ♦ One way to classify administrative tools is by use. The way that you plan to use the tool determines what features it requires. There are nine different tool types discussed in this chapter, each differing by usage: monitor, analysis, event, automated maintenance, manual maintenance, configuration, installation, user assistance, and tracking.
- ♦ Tools are also classified by where they're used and by who will use them. The two main considerations, in this case, are the level of administrator expertise and distance from the server. Less important, but of concern to the tool developer, are things such as the kind of data the tool will handle and whether the administrator needs to perform complex tasks with it. All of these factors determine the size and type of interface required by the tool.

- ♦ When determining what kind of tool to design, you have to consider various factors such as the anticipated longevity of the tool. In addition, it's important to look at the complexity of the environment. For example, do you really need to build an analysis tool instead of a simple monitor? Finally, it's important to consider the level of perceived automation needed versus the level that the administrator needs to maintain contact with the network.
- ♦ Always password protect the administrative tools you create if their use could cause irreparable damage to the network. The level of password protection should coincide with the amount of damage that could occur from unauthorized use.
- ♦ Using components is the easiest way to build administrative tools that work the same locally and through an Internet connection. DCOM provides the glue required to allow the component at the server communicate with the application requesting information.
- ♦ Security is a major problem for Internet-accessible administrative tools. The three levels of security threat include: unauthorized access, data interception or tampering, and outside disassembly of the tool's inner workings.
- ♦ Like most COM component types, MMC snap-ins support a variety of interfaces that help define the snap-in's capabilities. There are a variety of interfaces implemented by MMC that allow your snap-in to access resources and work with other snap-ins. Obviously, there are also MMC-specific interfaces that your component can support. The three most commonly implemented component interfaces are IComponent, IComponentData, and ISnapinAbout.
- ♦ The MMC SnapIn object type allows you to create new snap-ins with relative ease. However, it's important to select the right options within the ATL Object Wizard to ensure that the snap-in shell has all the features that you need at the outset; adding features later can be difficult.
- ♦ The RESULTDATAITEM data structure is central to item management within the Result View Pane of MMC.
- ♦ Never directly fill in the str data member for an item. Use the MMC_CALLBACK constant, and then fill in the string values on a column-by-column basis.
- ♦ The GetDisplayInfo() method is a required part of item presentation. You must place it within the class that implements the IComponent or IComponentImpl interface.
- ♦ Always define the contents of the String Table resource for your MMC snap-in component. These strings are used to define the object for the administrator. If the strings aren't defined, the administrator may not have any idea of what task your snap-in is capable of performing.



